KQ

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 957 428 A2

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
17.11.1999 Bulletin 1999/46

(51) Int Cl.6: G06F 9/38

(21) Application number: 99302988.3

(22) Date of filing: 19.04.1999

(84) Designated Contracting States:
AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE
Designated Extension States:
AL LT LV MK RO SI

(30) Priority: 12.05.1998 US 76593

(71) Applicant: International Business Machines Corporation
Armonk, NY 10504 (US)

(72) Inventor: McDonald, Robert Greg
Austin, Texas 78758 (US)

(74) Representative: Litherland, David Peter
IBM United Kingdom Limited
Intellectual Property Department
Hursley Park
Winchester, Hampshire SO21 2JN (GB)

(54) Method and apparatus for fetching noncontiguous instructions in a data processing system

(57) A method and apparatus for obtaining non-contiguous blocks of instruction in a data processing system is disclosed. The apparatus comprises an instruction cache means for providing a first plurality of instructions and branch logic means for receiving the first plurality of instructions and for providing branch history information about the first plurality of instructions. The apparatus further includes an auxiliary cache means for receiving a second plurality of instructions based upon the branch history information. The auxiliary cache means overlays at least a one of the second plurality of instructions if there is a branch in the first plurality of instructions and the branch is to the second plurality of instructions. Thus the apparatus can use branch history information and an auxiliary cache to fetch multiple noncontiguous groups of instructions in a single cycle. Furthermore, the technique allows noncontiguous fetching to be performed without requiring multiple levels of nested branch prediction logic to be evaluated in a single cycle.

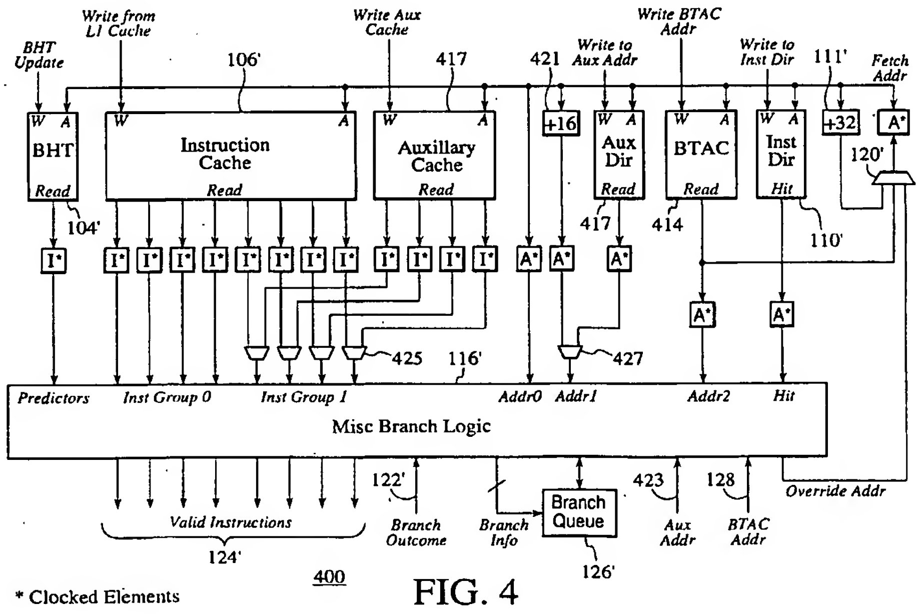

FIG. 4

EP 0 957 428 A2

Description

[0001] The present invention relates to a system and method for fetching noncontiguous instructions in a data processing system.

[0002] Superscalar processors employ aggressive techniques to exploit instruction-level parallelism. Wide dispatch and issue paths place an upper bound on peak instruction throughput. Large issue buffers are used to maintain a window of instructions necessary for detecting parallelism, and a large pool of physical registers provides destinations for all of the in-flight instructions issued from the window. To enable concurrent execution of instructions, the execution engine is composed of many parallel functional units. The fetch engine speculates past multiple branches in order to supply a continuous instruction stream to the window.

[0003] The trend in superscalar design is to scale these techniques: wider dispatch/issue, larger windows, more physical registers, more functional units, and deeper speculation. To maintain this trend, it is important to balance all parts of the processor-any bottlenecks diminish the benefit of aggressive techniques.

[0004] Instruction fetch performance depends on a number of factors. Instruction cache hit rate and branch prediction accuracy have been long recognized as important problems in fetch performance and are well-researched areas.

[0005] Because of branches and jumps, instructions to be fetched during any given cycle may not be in contiguous cache locations. Hence, there must be adequate paths and logic available to fetch and align noncontiguous basic blocks and pass them down the pipelines. That is, it is not enough for the instructions to be present in the cache, it must also be possible to access them in parallel.

[0006] Modern microprocessors routinely use Branch History Tables and Branch Target Address Caches to improve their ability to efficiently fetch past branch instructions. Branch History Tables and other prediction mechanisms allow a processor to fetch beyond a branch instruction before the outcome of the branch is known. Branch Target Address Caches allow a processor to speculatively fetch beyond a branch before the branch's target address has been computed. Both of these techniques use run-time history to speculatively predict which instructions should be fetched and eliminate "dead" cycles that might normally be wasted. Even with these techniques, current microprocessors are limited to fetching only contiguous instructions during a single clock cycle.

[0007] As superscalar processors become more aggressive and attempt to execute many more instructions per cycle, they must also be able to fetch many more instructions per cycle. Frequent branch instructions can severely limit a processor's effective fetch bandwidth. Statistically, one of every four instructions is a branch instruction and over half of these branches are taken. A processor with a wide fetch bandwidth, say 8 contiguous instructions per cycle, could end up throwing away half of the instructions that it fetches as much as half of the time.

[0008] High performance superscalar processor organizations divide naturally into an instruction fetch mechanism and an instruction execution mechanism. The fetch and execution mechanisms are separated by instruction issue buffer(s), for example, queues, reservation stations, etc. Conceptually, the instruction fetch mechanism acts as a "producer" which fetches, decodes, and places instructions into the buffer. The instruction execution engine is the "consumer" which removes instructions from the buffer and executes them, subject to data dependence and resource constraints. Control dependences (branches and jumps) provide a feedback mechanism between the producer and consumer.

[0009] Previous designs use a conventional instruction cache, containing a static form of the program, to work with. Every cycle, instructions from noncontiguous locations must be fetched from the instruction cache and assembled into the predicted dynamic sequence. There are problems with this approach:

[0010] Pointers to all of the noncontiguous instruction blocks must be generated before fetching can begin. This implies a level of indirection, through some form of branch target table (branch target buffer, branch address cache, etc.), which translates into an additional pipeline stage before the instruction cache.

[0011] The instruction cache must support simultaneous access to multiple, noncontiguous cache lines. This forces the cache to be multiported: if multiporting is done through interleaving, bank conflicts are suffered.

[0012] After fetching the noncontiguous instructions from the cache, they must be assembled into the dynamic sequence. Instructions must be shifted and aligned to make them appear contiguous to the decoder. This most likely translates into an additional pipeline stage after the instruction cache.

[0013] A trace cache approach avoids these problems by caching dynamic sequences themselves, ready for the decoder. If the predicted dynamic sequence exists in the trace cache, it does not have to be recreated on the fly from the instruction cache's static representation. In particular, no additional stages before or after the instruction cache are needed for fetching noncontiguous instructions. The stages do exist, but not on the critical path of the fetch unit-rather, on the fill side of the trace cache. The cost of this approach is redundant instruction storage: the same instructions must reside in both the primary cache and the trace cache, and there even might be redundancy among lines in the trace cache. Accordingly, utilizing a trace cache approach several instructions are grouped together based upon a most likely path. They are then stored together in the trace cache. This system requires a complex mechanism to pack and cache instruction segments.

[0014] Accordingly, a need exists for a technique for improving the overall throughput of a superscalar processor. More particularly, what is needed is a system and method for efficiently fetching noncontiguous instructions in such a processor.

[0015] A method and system for obtaining non-contiguous blocks of instruction in a data processing system is disclosed. In a first aspect, apparatus for fetching noncontiguous blocks of instructions in a data processing system is disclosed. The apparatus comprises an instruction cache means for providing a first plurality of instructions and branch logic means for receiving the first plurality of instructions and for providing branch history information about the first plurality of instructions. The apparatus further includes an auxiliary cache means for receiving a second plurality of instructions based upon the branch history information. The auxiliary cache means overlays at least a one of the second plurality of instructions if there is a branch in the first plurality of instructions and the branch is to the second plurality of instructions.

[0016] In a second aspect, a method for obtaining noncontiguous blocks of instruction comprises storing a first plurality of instructions in a first cache and fetching the first plurality of instructions in parallel with a fetch of a second plurality of instructions within a second cache. In the present invention, the number of the second plurality of instructions being greater than the number of first plurality of instructions. This second aspect includes replacing a portion of the second plurality of instructions with at least one of the first plurality of instructions based upon a branch history information of the data processing system.

[0017] The above-described present invention allows a processor to use branch history information and an auxiliary cache to fetch multiple noncontiguous groups of instructions in a single cycle. Furthermore, the technique allows noncontiguous fetching to be performed without requiring multiple levels of nested branch prediction logic to be evaluated in a single cycle.

[0018] An embodiment of the invention will now be described, by way of example only, with reference to the accompanying drawings in which:

Figure 1 is a block diagram of a superscalar processor;

Figure 2A is a block diagram of a conventional mechanism within a processor for fetching noncontiguous instructions;

Figure 2B is a block diagram of an instruction cache and branch target address cache entry;

Figure 3 is a flow chart of a branch prediction algorithm for the conventional mechanism of Figure 2;

Figure 4 is a block diagram of a mechanism within a processor for fetching noncontiguous instructions in a single cycle in accordance with the embodiment of the present invention;

Figure 5 is a flow chart of the branch prediction algorithm for the noncontiguous instruction fetch mechanism of Figure 5; and

Figure 6 is a table that illustrates the flow of instructions when utilizing the branch prediction algorithm of Figure 5.

[0019] Figure 1 is a block diagram of a superscalar processor 10. As shown, the superscalar processor 10 typically include a system bus 11 connected to a bus interface unit ("BIU") 12. BIU 12 controls the transfer of information between processor 10 and system bus 11. BIU 12 is connected to an instruction cache 14 and to a data cache 16 of processor 10. Instruction cache 14 outputs instructions to a sequencer unit 18. In response to such instructions from instruction cache 14, sequencer unit 18 selectively outputs instructions to other execution circuitry of processor 10.

[0020] In addition to sequencer unit 18 which includes execution units of a dispatch unit 46 and a completion unit 48, in the preferred embodiment the execution circuitry of processor 10 includes multiple execution units, namely a branch unit 20, a fixed point unit A ("FXUA") 22, a fixed point unit B ("FXUB") 24, a complex fixed point unit ("CFXU") 26, a load/store unit ("LSU") 28 and a floating point unit ("FPU") 30. FXUA 22, FXUB 24, CFXU 26 and LSU 28 input their source operand information from general purpose architectural registers ("GPRs") 32 and fixed point rename buffers 34. Moreover, FXUA 22 and FXUB 24 input a "carry bit" from a carry bit ("CA") register 42. FXUA 22, FXUB 24, CFXU 26 and LSU 28 output results (destination operand information) of their operations for storage at selected entries in fixed point rename buffers 34. Also, CFXU 26 inputs and outputs source operand information and destination operand information to and from special purpose registers ("SPRs") 40.

[0021] FPU 30 inputs its source operand information from floating point architectural registers ("FPRs") 36 and floating point rename buffers 38. FPU 30 outputs results (destination operand information) of its operation for storage at selected entries in floating point rename buffers 38.

Processes

[0022] The processor 10 is usually implemented with a large number of state machines that control relatively independent processes. It may be thought of as a complex parallel algorithm involving multiple concurrent processes.

Instruction Fetch

[0023] This process provides a continuous stream of instructions from the instruction cache and utilizes the branch target address cache (BTAC) as a fetch prediction mechanism.

Branch Prediction

[0024] This process identifies and predicts branches, verifies that the appropriate instructions were fetched, updates information, and places information about speculative branches into a Branch Queue.

Branch Resolution

[0025] This process checks that the prediction matched the actual branch result and makes corrections if a misprediction occurred.

Branch Completion

[0026] This process writes information for completed branches into the BHT and removes entries from the Branch Queue.

[0027] The present embodiment relates generally to the fetch cycle and the ability to fetch noncontiguous instructions. Figure 2A shows in system 100 the hardware mechanisms for the conventional technique for fetching groups of instructions. In this embodiment, eight instructions are shown being fetched at a time. In addition, the present invention is discussed with reference to instructions being four bytes in length. However, one of ordinary skill in the art, readily recognizes than any number of instructions can be fetched at a time and can be of any length and that number and length would be within the scope of the present invention.

[0028] Referring back to Figure 2A, as is seen there is a fetch address signal 102 which is provided to a branch history table (BHT) 104, an instruction cache 106, a branch target address cache (BTAC) 108, a directory for the instruction cache (INST Dir) 110, +32 counter 111. To provide a fuller understanding of the operation of the above-identified mechanism, particularly as it relates to the instruction cache 106, a conventional instruction cache entry is described below.

[0029] Figure 2B shows a simple organization for the instruction cache and BTAC entry 200 of the instruction required by an instruction fetcher (this entry may include other information that is not important for this discussion). Figure 2B shows a fetch information 201 which includes a sample address tag 202, successor index 204, and branch block index entries 206 for a code sequence, assuming a 64-Kbyte, direct-mapped cache and the indicated instruction addresses. For this example, the cache entry holds four instructions 208, 210, 212 and 214. The entry also contains instruction-fetch information. The fetch information also includes two additional fields (not shown) used by the instruction fetcher.

[0030] The successor index field 204 indicates both the next cache block predicted to be fetched and the first instruction within this next block predicted to be executed. The successor index field 204 does not specify a full instruction address, but is of sufficient size to select any instruction within the cache. For example, a 64-Kbyte, direct-mapped cache requires a 14-bit successor index if all instructions are 32 bits in length (12 bits to address the cache block and 2 bits to address the instruction in the block if the block size is four words).

[0031] In a preferred embodiment, the branch block index field 206 indicates the location of a branch point within the corresponding instruction block. Instructions beyond the branch point are predicted not to be executed.

[0032] Referring back to Figure 2A, the BHT 104 also receives a BHT update signal and outputs a read signal. The read signal from the BHT 104 is provided to branch logic 116. The instruction cache 106 receives write signal from an outside source, such as an L2 cache. The instruction cache 106 outputs eight instructions (instruction group 0) to the branch logic 116. An address 0 signal is provided directly to the branch logic 116. The branch logic 116 provides an override address signal to multiplexer 120. Multiplexer 120 also receives signals 32, counter 111 and the output of BTAC 102. An address 1 signal is provided from BTAC 108 to branch logic 116. The instruction directory 110 provides a hit signal to the branch logic 116. The branch logic 116 also receives the branch outcome signal, provides branch information to a branch queue 1 216, and outputs a BTAC address 128 and provides valid instructions 124. This type of mechanism is capable of fetching eight contiguous instructions per cycle, but would only use instructions up to the first predicted taken branch in the group. To explain this in more detail refer to the following discussion in combination with the accompanying figures.

[0033] As before described, there are several processes associated with the fetching of groups of instructions. The present invention is related to an improvement in the branch prediction algorithm and an associated modification to the conventional fetch mechanism of Figure 2A.

[0034] To further illustrate the problems associated with the fetching of non-contiguous instructions with regard to the conventional mechanisms of Figure 2A, refer now to Figure 3.

[0035] Figure 3 is a flow chart of a branch prediction algorithm for the conventional mechanism of Figure 2A. Referring to Figures 2A and 3 together, first it is determined whether valid instructions are found in the instruction cache, via step 302. If there are no valid instructions found in the instruction cache, then all fetched instructions are invalidated and the miss handler is initiated, via step 304. However, if there are valid instructions found in the instruction cache, then branches are identified, target addresses are computed, and predicted taken or not taken based on the branch logic 116 and the branch history tables 104, via step 306. Thereafter it is determined whether there is a predicted taken branch in instruction group 0 (the first group of instructions), via step 308. If there is a predicted branch taken, then all subsequent instructions are invalidated, via step 310. Then it is determined if the address 1 (address of the second group of instructions) from BTAC 108 is equal to target 0 of the instruction directory, via step 312. If the answer is yes, then the branch addresses are stored and prediction information for all branches are provided into the branch queue 126, via step 314. If, on the other hand, address 1 is not equal to target 0, then the instructions that have been fetched in the next cycle are invalidated and the override address is equal to target 0, via step 316. Thereafter, the BTAC address is updated to equal target 0, via step 318, and then the branch addresses and predicted information is stored into the branch queue via step 314. If, on the other hand, at step 308 there are no predicted branches taken in group 0, then it is determined if address 1 is equal to address 0 plus 32. If the answer is yes, then return to step 314. If, on the other hand, the answer is no, then all instruction groups fetched for the next cycle are invalidated and the override address equals address 0 + 32, via step 322. Thereafter, the BTAC address is updated to equal invalid, via step 324 and then return to step 314.

[0036] This algorithm of Figure 3 does not allow for fetching non-contiguous instructions in a single cycle. This prediction algorithm always requires that when a branch instruction is encountered only the instructions up to the branch can be utilized. As has been beforementioned, there are mechanisms, i.e., trace cache, etc., to retrieve noncontiguous instructions in a single cycle but they add complexity and cost to the system.

[0037] The present invention overcomes this problem by providing an auxiliary cache and an overlaying technique that utilizes the auxiliary cache to fetch noncontiguous instructions in a single cycle.

[0038] In the present embodiment, three major hardware mechanisms are required for this technique:

(1) a Standard Instruction Cache (or other memory source),

(2) a Branch Target Address Cache, and

(3) an auxiliary cache.

[0039] A Standard Instruction Cache and a Branch Target Address Cache are commonly used in most microprocessors and may be used without modification for this technique. The auxiliary cache is a new hardware mechanism that contains multiple entries with one or more instructions and an associated address. The auxiliary cache may be highly associative and relatively small compared to the main instruction cache.

[0040] The present embodiment operates generally in the following manner:

1. If a branch instruction in the first instruction group is considered strongly taken (based on branch history or other information) and no instructions were provided from the auxiliary cache, use the fetch index to add the branch's target address and one or more instructions at that address to the auxiliary cache. Also, an appropriate sequential address is provided as needed (e.g., branch target plus 16 bytes) to the BTAC.

2. Else, if a branch instruction in the second instruction group is considered strongly taken, use the fetch index to add the branch's target address to the BTAC.

3. Else, if no branch instructions in either instruction group is considered strongly taken, use the fetch index to clear the BTAC and default to an appropriate sequential address.

[0041] To more particularly describe the features and operation of the present embodiment, refer now to the following discussion in conjunction with the accompanying figures.

[0042] Figure 4 is a block diagram of a mechanism 400 within a processor for fetching noncontiguous instructions in a single cycle in accordance with the present invention. The elements of the mechanism 400 are similar to many of the elements presently in mechanism 100. Those elements that are similar have been given similar reference numerals. As has been before mentioned, the key element that is different is the addition of the auxiliary cache 415 and its directory 417.

[0043] In addition, as is seen, there are instruction groups 0 and 1 and as is seen, there are four multiplexers 425 which allow for the overlaying of the instructions from the auxiliary cache 417 in the instruction group 1 from the instruction cache 106' based on branch history information derived from BHT 104' and branch logic 116'. Similarly, the auxiliary directory also overlays its address over the address 1 signal of the +16 counter 421 based upon the branch history information. In addition, FTAC 419 also provides an address 2 signal rather than the address 1 signal provided by the BTAC 108' of Figure 2A. Accordingly, as has been before mentioned, through the addition of the auxiliary cache 415 and the use of it and the auxiliary directory 417, it is possible now to accumulate information to allow for fetching of noncontiguous instructions. To further describe this feature in a more detailed manner, refer now to Figure 5.

[0044] Figure 5 is a flow chart of the branch prediction algorithm for the noncontiguous instruction fetch mechanism of Figure 4. Referring now to Figures 4 and 5 together, first it is determined whether there are valid instructions stored in the instruction cache 106', via step

502. If there were no valid instructions found in the instruction cache 106', then all instructions are invalidated and a miss handler is initiated, via step 504. If, on the other hand, there are valid instructions in the instruction cache 106', it is next determined whether there were valid instructions found in the auxiliary cache 417, via step 506. If there were valid instructions found in the auxiliary cache 417, then the instructions from the auxiliary cache 417 are overlaid on the instruction group from the instruction cache 106', via step 508. If, on the other hand, there were no valid instructions found in the auxiliary cache, then all the instructions from the instruction cache are retained, via step 510.

[0045] From either of the steps 508 and 510, next branches are identified, target addresses are computed, and they are predicted taken or not taken, via step 512 based upon the branch logic 106' and the BHT 104' operating in a conventional manner. Thereafter, it is then determined if there is a predicted branch taken in the instruction group 0, via step 514. If there is a predicted taken branch in instruction group 0, then the subsequent instructions are invalidated, via step 516.

[0046] Next, it is determined whether address 1 is equal to the target address 0 of the branch, via step 518. If the answer is yes, then it is determined whether there is a predicted taken branch in group 1 or in the next group of instructions, via step 520. If the answer to that question is yes, then the subsequent instructions in instruction group 1 are invalidated, via step 522. Then it is determined whether address 2 is equal to target 1 of the branch, via step 524. If the answer to that is yes, then the branch queue stores the branch addresses and prediction information for all the branches, via step 526. If the answer is no, then the next cycle group is invalidated and the override address equals target 1, via step 528. Thereafter, the FTAC address is updated to equal target 1, via 530, and return to step 526.

[0047] If address 1 is not equal to target 0 via step 518, then all the instructions in group 1 and the next cycle groups are invalidated and the override address equals target 0 and prepare to save next group in auxiliary cache, via step 521. Thereafter, the auxiliary address is updated to equal target 0 and the FTAC address equals target 0 + 16, via step 523.

[0048] If, on the other hand, there is no predicted branch taken in group 1, via step 520, then it is then determined if address 2 is equal to address 1 plus 16, via step 532. If the answer is yes, then return to step 526. If the answer is no, then all the next cycle groups are invalidated and the override address set to equal address 1 + 16, via step 534. Thereafter the FTAC address is updated to equal address 1 + 16, via step 536, and then return to step 526.

[0049] Returning now to step 514, if there is no predicted branches taken in group 0, then it is determined if address 1 equals address 0 + 16. If address 1 is equal to address 0 + 16, then return to step 520 and proceed through the steps based on that decision chain. If, on the other hand, the answer is no, that address 1 does not equal address 0 + 16, then all instructions in group 1 and the next cycle groups are invalidated and the override address equals address 0 + 16, via step 540. Thereafter, the auxiliary address is updated to equal invalid and the FTAC address is updated to equal invalid, via step 542. Then return to step 526. Accordingly, through this branch prediction process, the system can accumulate branch history information in a manner to allow for the overlay instruction of auxiliary cache to efficiently fetch a noncontiguous instructions. To more clearly describe the operation in the context of a particular example, refer now to Figure 6.

[0050] Figure 6 is an example 600 that illustrates the flow of instructions when utilizing the branch prediction algorithm of Figure 5. The example 600 shown in Figure 6 illustrates a sequence of fetches 602 performed on consecutive cycles in accordance with the present invention for a program segment 604. Note that all the addresses are described using a hexadecimal format (base 16). Asterisks in the example 600 indicate fetched instructions that were invalidated from the instruction stream.

[0051] As is seen, the program segment comprises a plurality of basic blocks 606, 608, 610 and 612. Each of the basic blocks 606-612 begin with a load instruction and end with a branch instruction. The basic blocks are used in conjunction with the present invention to allow for non-contiguous instructions to be obtained in a single cycle.

[0052] Through the use of the branch prediction algorithm of Figure 5 in conjunction with the hardware mechanisms of Figure 4, to accumulation branch history information, noncontiguous instructions of Figure 6 can be obtained in a single cycle.

[0053] To illustrate the method for obtaining instructions in the single cycle, refer to Figures 4, 5 and 6. As has been indicated, at cycle 000 there are eight instructions being provided. It is assumed that the auxiliary cache 417 initially contains no instructions, and so at this time there are invalid instructions found in the information cache 502, then it is determined whether valid instructions are found in the auxiliary cache 506, and the answer would be no. In that event, then all the instructions from the instruction cache would be retained. At that point, branches are identified, target addresses are computed, and predicted taken or not taken, via step 512. It is known that the target for the branch in the first set of instructions is at 0x100. It is now determined whether there is a predicted branch taken in group 0, and the answer to that is yes. That is the third instruction in address 000. Through the branch prediction process the instruction at address 100 is provided to the auxiliary and the address is stored in the auxiliary directory. Also, through the branch prediction process address 110 would be stored in the BTAC 108 (step 530).

[0054] Then the next basic block 608 is used at cycle 003 to load the instructions at the target address 0x100.

The last instructions at basic block 608 is a branch to address 200. Then address 100, as before mentioned, is then fetched and similar information is accumulated for it in the auxiliary cache, auxiliary directory and BTAC. Accordingly, informatión is accumulated in cycles 003-007 until, as is seen in cycle 008, two non-contiguous instructions (0X000 and 0X100) are retrieved in a single cycle.

[0055] This branch prediction process is repeated again, via basic blocks 610 and 612, wherein non-contiguous instructions are fetched in cycles 020 through 024. Accordingly, as is seen in this example, after branch history information is accumulated in a sufficient manner, then noncontiguous instructions can be obtained in a single cycle. This process can be repeated, particularly in those instances where instructions recur, such that a majority of noncontiguous instructions can be retrieved in a single cycle. This is accomplished through the use of the auxiliary cache and the miscellaneous branch logic cooperating with the branch history tables while utilizing the branch prediction process in accordance with the present invention.

[0056] Several other techniques, such as trace caching and multi-level branch predictors; have been proposed for allowing a processor to fetch noncontiguous instructions in a single cycle. The auxiliary cache and instruction overlay technique described above is simpler than other techniques, but as effective.

[0057] Although the present invention has been described in accordance with the embodiments shown, one of ordinary skill in the art will readily recognize that there could be variations to the embodiments and those variations would be within the spirit and scope of the present invention. Accordingly, many modifications may be made by one of ordinary skill in the art without departing from the scope of the appended claims.

Claims

1. Apparatus for fetching noncontiguous blocks of instructions in a data processing system; the system comprising:

   an instruction cache means for providing a first plurality of instructions;

   branch logic means for receiving the first plurality of instructions and for providing branch history information about the first plurality of instructions; and

   an auxiliary cache means for receiving a second plurality of instructions based upon the branch history information; the auxiliary cache means overlaying at least a one of the second plurality of instructions if there is a branch in the first plurality of instructions and the branch is to the second plurality of instructions.

2. Apparatus as claimed in claim 1 in which the auxiliary cache means comprises an auxiliary cache and an auxiliary directory.

3. Apparatus as claimed in claim 1 or claim 2 in which the first plurality of instructions comprises two blocks of instructions.

4. Apparatus as claimed in any preceding claim in which the second plurality of instructions comprises one block of instructions.

5. Apparatus as claimed in any preceding claim which further comprises a branch address target cache coupled to the branch logic.

6. A method for obtaining non-contiguous blocks of instructions in a data processing system; the method comprising the steps of:

   (a) storing a first plurality of instructions in a first cache;

   (b) fetching the first plurality of instructions in parallel with a fetch of a second plurality of instructions within a second cache, the number of the second plurality of instructions being greater than the number of the first plurality of instructions; and

   (c) replacing a portion of the second plurality of instructions with at least one of the first plurality of instructions based upon a branch history information of the data processing system.

7. The method of claim 6 wherein the first cache comprises an instruction cache and the second cache comprises an auxiliary cache.

8. The method of claim 7 in which the auxiliary cache includes an auxiliary directory.

9. The method of any of claims 6 to 8 which the second plurality of instructions comprises two blocks of instructions.

10. The method of any of claims 6 to 9 in which the first plurality of instructions comprises one block of instructions.

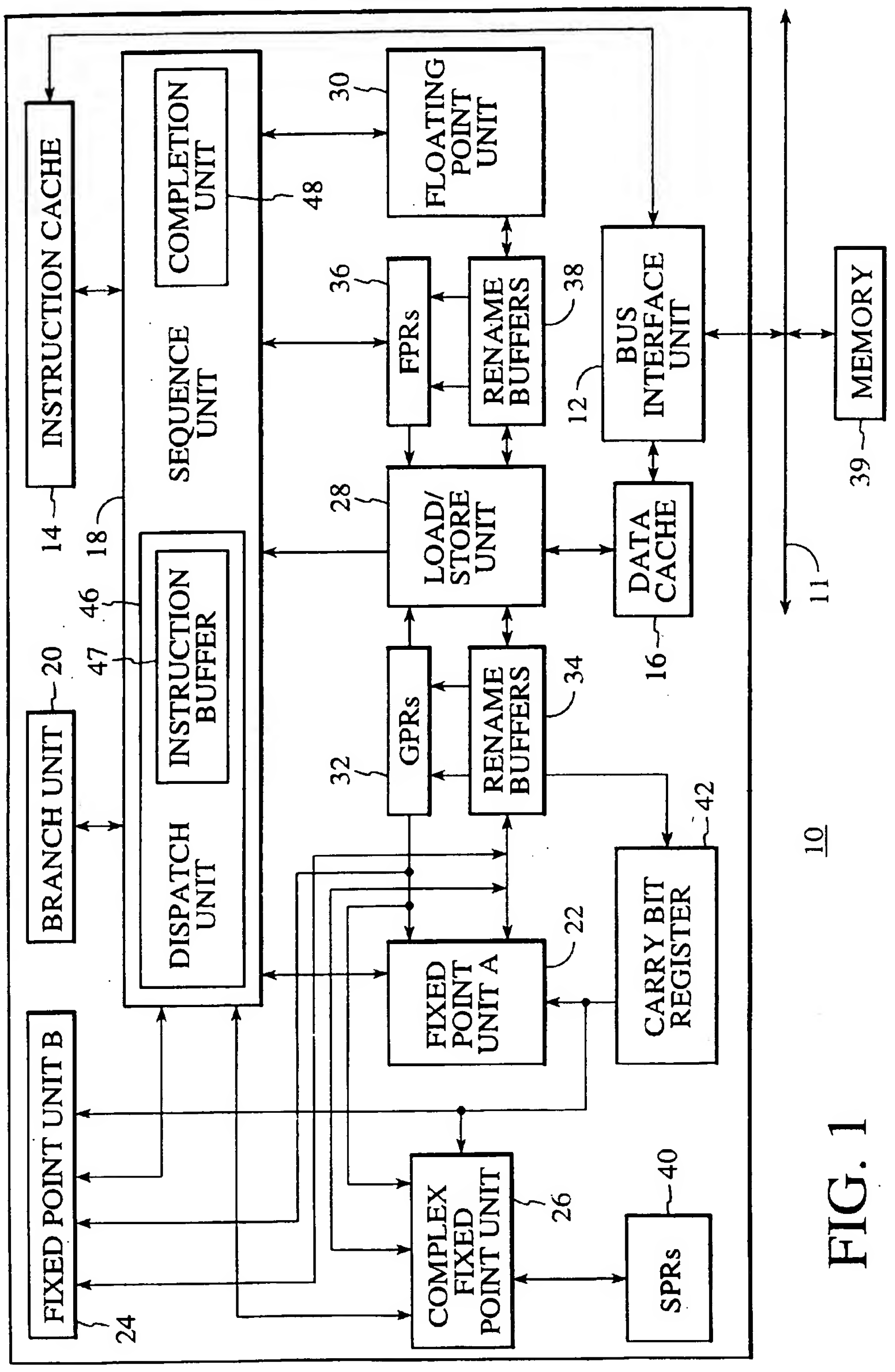
INSTRUCTION CACHE
14
SEQUENCE UNIT
18
COMPLETION UNIT
48
INSTRUCTION BUFFER
47
46
DISPATCH UNIT
BRANCH UNIT
20
FIXED POINT UNIT B
24
FLOATING POINT UNIT
30
FPRs
36
RENAME BUFFERS
38
LOAD/ STORE UNIT
28
GPRs
32
RENAME BUFFERS
34
FIXED POINT UNIT A
22
COMPLEX FIXED POINT UNIT
26
BUS INTERFACE UNIT
12
DATA CACHE
16
CARRY BIT REGISTER
42
SPRs
40
MEMORY
39
11
10

FIG. 1

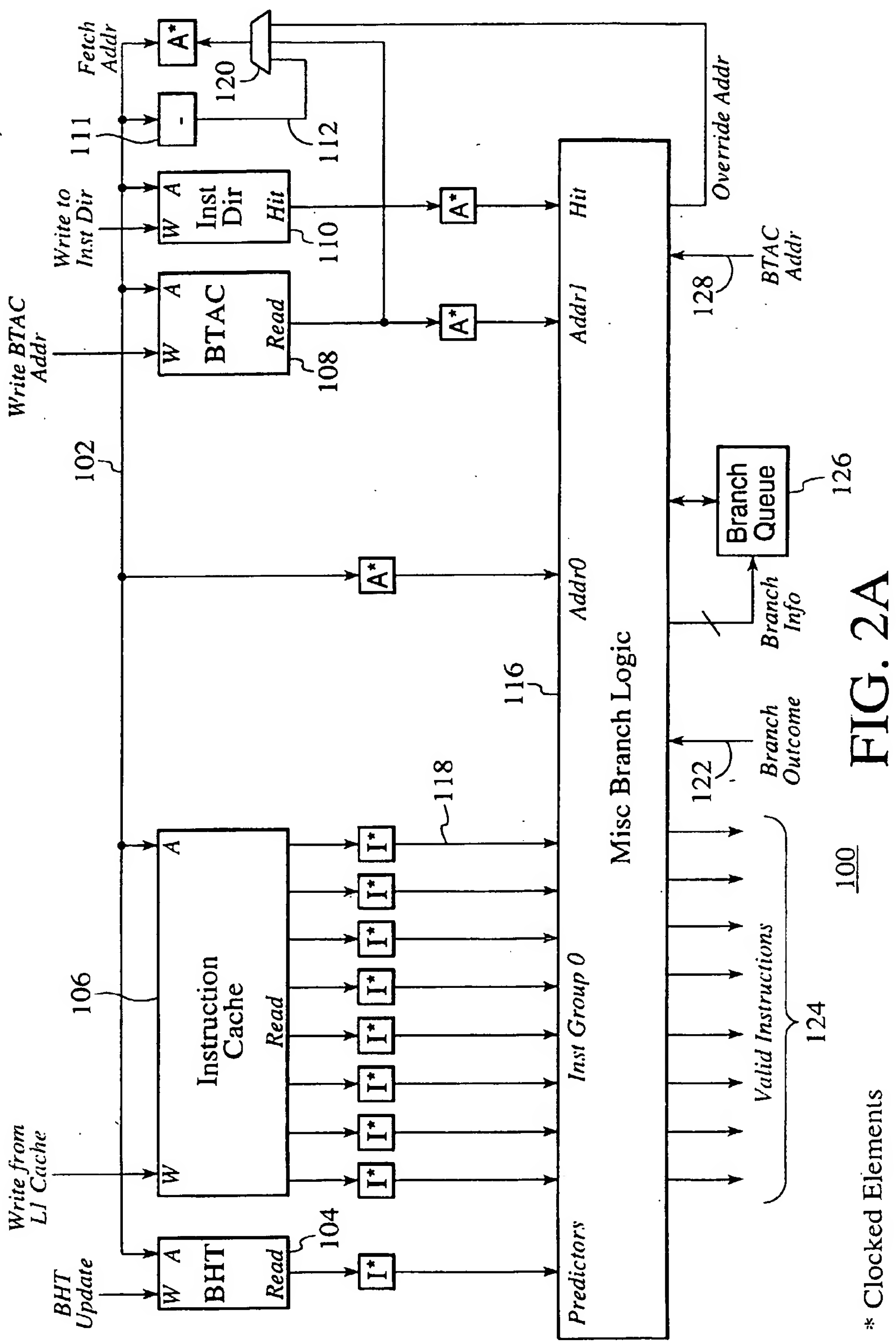
Fetch Addr
A*
120
111
112
Write to Inst Dir
W A Inst Dir Hit
110
Write BTAC Addr
W A BTAC Read
108
102
A*
A*
A*
Override Addr
Hit
Addr1
Addr0
BTAC Addr
128
Branch Queue
126
Branch Info
116
Misc Branch Logic
Branch Outcome
122
118
106
Instruction Cache
Read
A
W
Write from L1 Cache
I*
Inst Group 0
Valid Instructions
124
100
BHT Update
W A BHT Read
104
Predictors
* Clocked Elements

FIG. 2A

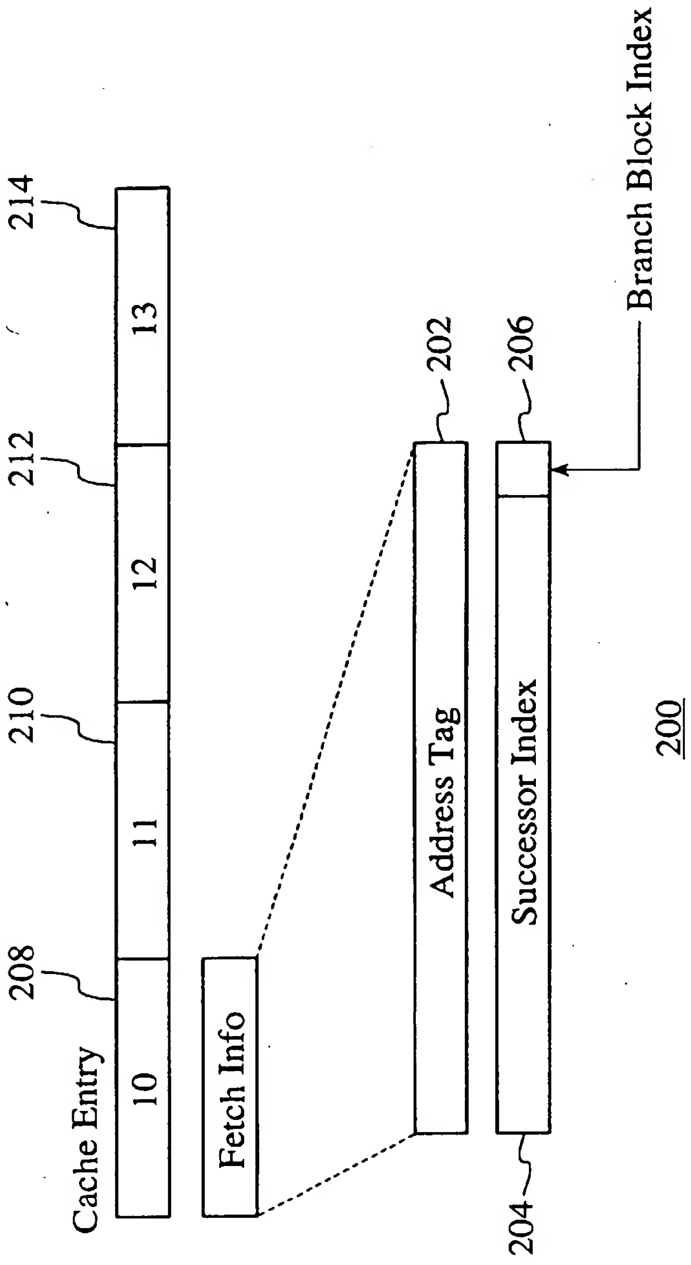
Cache Entry
10
11
12
13
208
210
212
214
Fetch Info
Address Tag
Successor Index
202
204
206
Branch Block Index
200

FIG. 2B

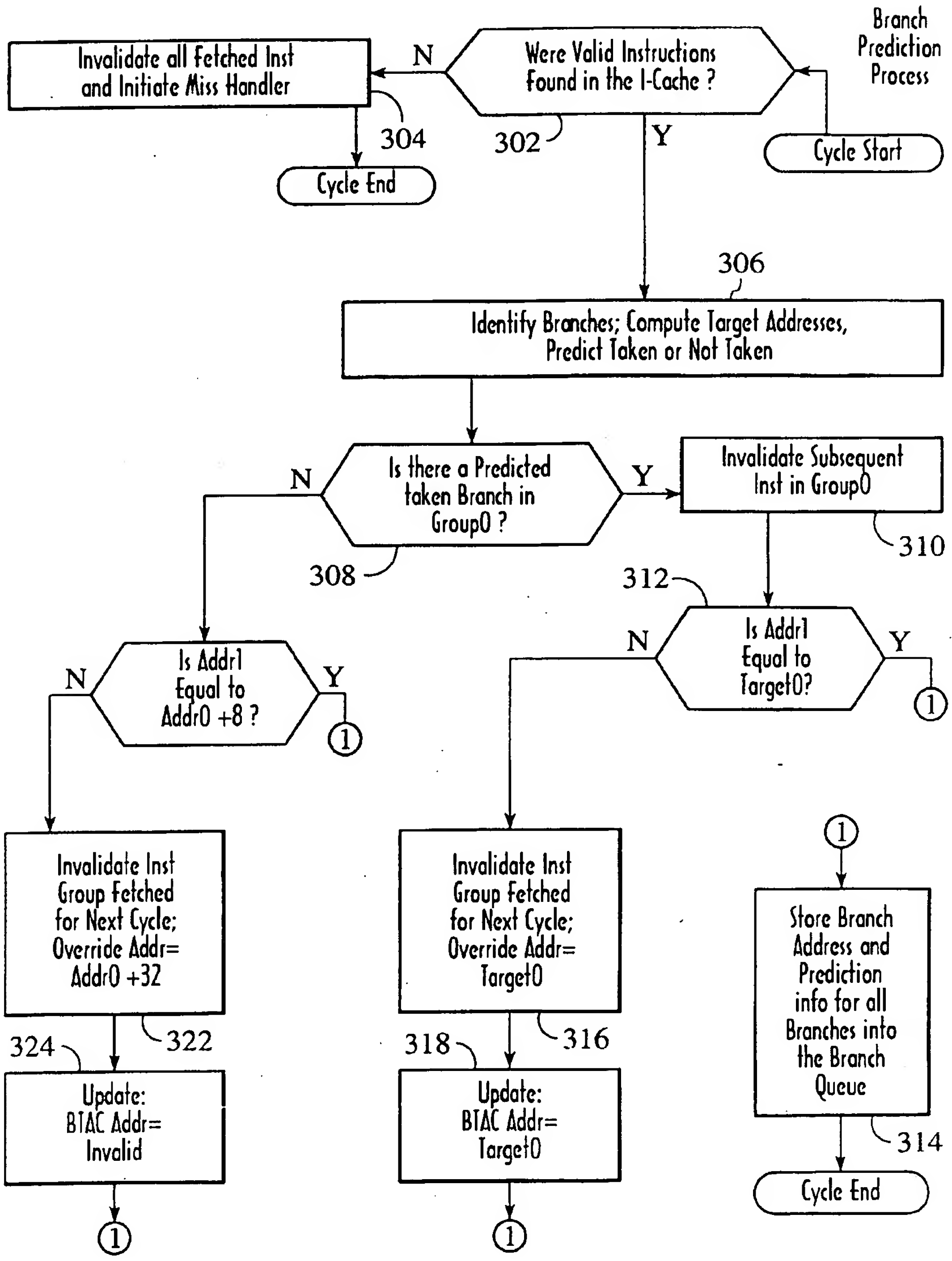
Branch Prediction Process
Cycle Start
Were Valid Instructions Found in the I-Cache ?
302
N
Y
Invalidate all Fetched Inst and Initiate Miss Handler
304
Cycle End
306
Identify Branches; Compute Target Addresses, Predict Taken or Not Taken
Is there a Predicted taken Branch in Group0 ?
308
N
Y
Invalidate Subsequent Inst in Group0
310
312
Is Addr1 Equal to Target0?
N
Y
1
Is Addr1 Equal to Addr0 +8 ?
N
Y
1
Invalidate Inst Group Fetched for Next Cycle; Override Addr= Addr0 +32
322
324
Update: BTAC Addr= Invalid
1
Invalidate Inst Group Fetched for Next Cycle; Override Addr= Target0
316
318
Update: BTAC Addr= Target0
1
1
Store Branch Address and Prediction info for all Branches into the Branch Queue
314
Cycle End

FIG. 3

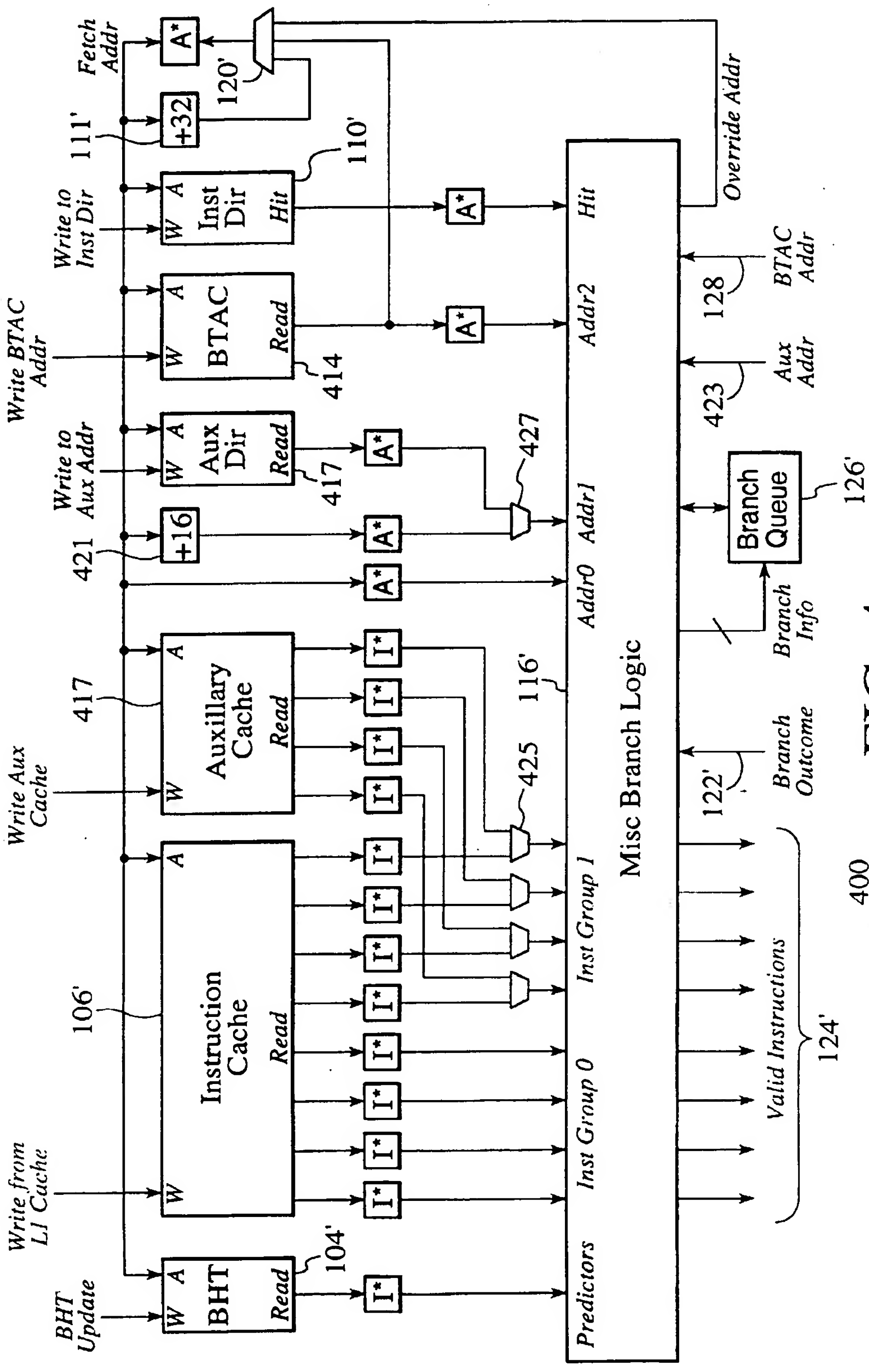

FIG. 4

* Clocked Elements

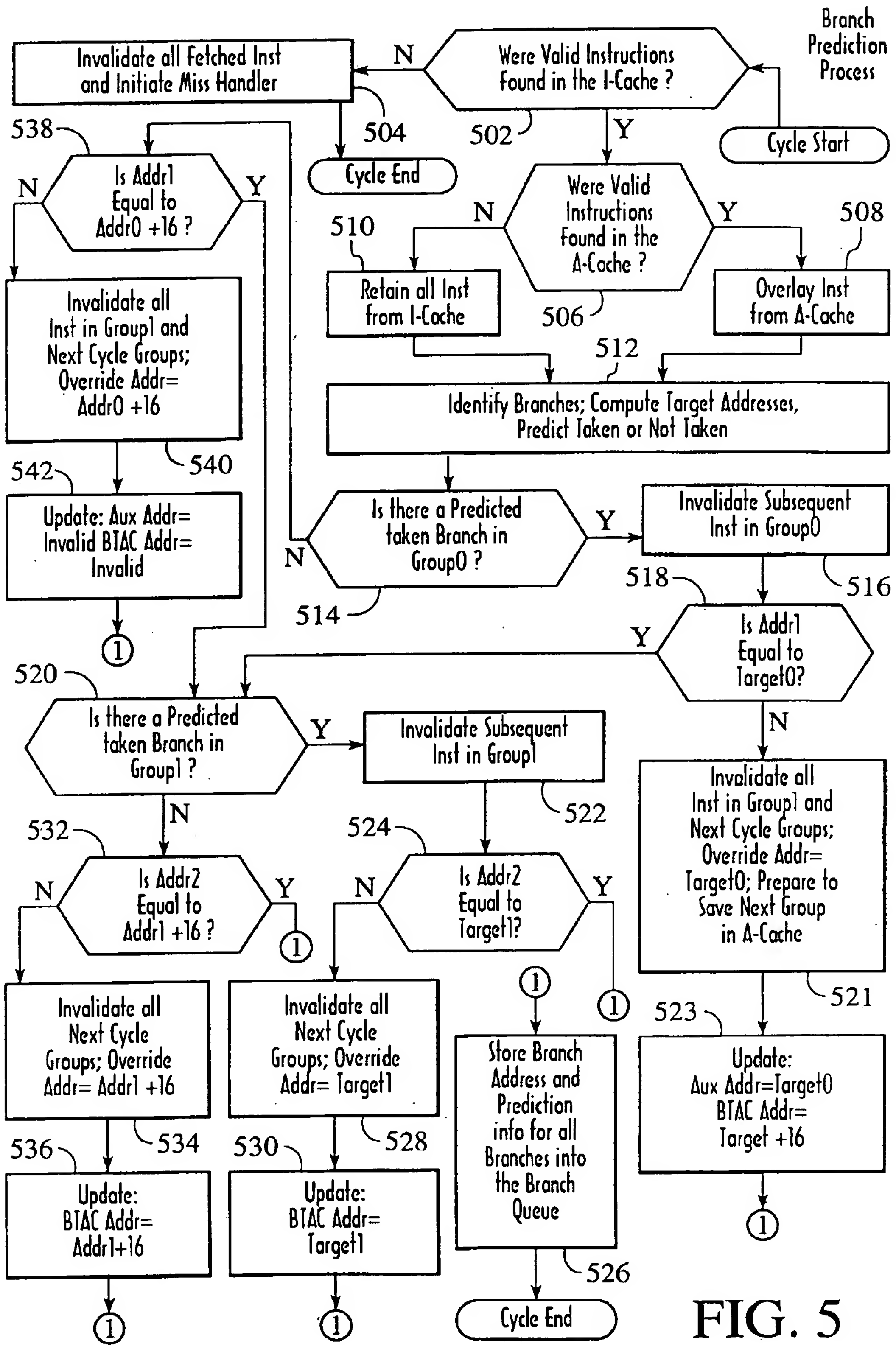
Branch Prediction Process
Cycle Start
Were Valid Instructions Found in the I-Cache ?
N
Y
502
Invalidate all Fetched Inst and Initiate Miss Handler
504
Cycle End
Were Valid Instructions Found in the A-Cache ?
506
N
Y
510
Retain all Inst from I-Cache
508
Overlay Inst from A-Cache
512
Identify Branches; Compute Target Addresses, Predict Taken or Not Taken
Is there a Predicted taken Branch in Group0 ?
514
N
Y
Invalidate Subsequent Inst in Group0
516
518
Is Addr1 Equal to Target0?
Y
N
Invalidate all Inst in Group1 and Next Cycle Groups; Override Addr= Target0; Prepare to Save Next Group in A-Cache
521
523
Update: Aux Addr=Target0 BTAC Addr= Target +16
1
538
Is Addr1 Equal to Addr0 +16 ?
N
Y
Invalidate all Inst in Group1 and Next Cycle Groups; Override Addr= Addr0 +16
540
542
Update: Aux Addr= Invalid BTAC Addr= Invalid
1
520
Is there a Predicted taken Branch in Group1 ?
Y
N
Invalidate Subsequent Inst in Group1
522
532
Is Addr2 Equal to Addr1 +16 ?
N
Y
1
Invalidate all Next Cycle Groups; Override Addr= Addr1 +16
534
536
Update: BTAC Addr= Addr1+16
1
524
Is Addr2 Equal to Target1?
N
Y
1
1
Invalidate all Next Cycle Groups; Override Addr= Target1
528
530
Update: BTAC Addr= Target1
1
Store Branch Address and Prediction info for all Branches into the Branch Queue
526
Cycle End

FIG. 5

FIG. 6

600

604

Program Segment

	Addr	Inst
606	0x000	Load
606	0x004	Compare
606	0x008	Br 0x100
	.....	.....
608	0x100	Load
608	0x104	Add
608	0x108	Store
608	0x10c	Br 0x200
	.....	.....
610	0x200	Load
610	0x204	Load
610	0x208	Add
610	0x20c	Store
610	0x210	Compare
610	0x214	Br 0x300
	.....	.....
612	0x300	Load
612	0x304	Add
612	0x308	Store
612	0x30c	Br 0x000

602

Sample Sequence of Fetched Instructions

Cycle	Addr0	Group0	Addr1	Group1	Addr2
000	0x000	L C B *	0x010	* * * *	0x020
001	0x020	* * * *	0x030	* * * *	0x040
002	0x100	L A S B	0x110	* * * *	0x120
003	0x120	* * * *	0x130	* * * *	0x140
004	0x200	L L A S	0x210	C B * *	0x220
005	0x220	* * * *	0x230	* * * *	0x240
006	0x300	L A S B	0x310	* * * *	0x320
007	0x320	* * * *	0x330	* * * *	0x340
008	0x000	L C B *	0x100	L A S B	0x110
009	0x110	* * * *	0x120	* * * *	0x130
010	0x200	L L A S	0x210	C B * *	0x300
011	0x300	L A S B	0x000	L C B *	0x010
012	0x010	* * * *	0x020	* * * *	0x030
013	0x100	L A S B	0x200	L L A S	0x210
014	0x210	C B * *	0x220	* * * *	0x230
015	0x230	* * * *	0x240	* * * *	0x250
016	0x300	L A S B	0x000	L C B *	0x100
017	0x100	L A S B	0x200	L L A S	0x210
018	0x210	C B * *	0x300	L A S B	0x310
019	0x310	* * * *	0x320	* * * *	0x330
020	0x000	L C B *	0x100	L A S B	0x200
021	0x200	L L A S	0x210	C B * *	0x300
022	0x300	L A S B	0x000	L C B *	0x100
023	0x100	L A S B	0x200	L L A S	0x210
024	0x210	C B * *	0x300	L A S B	0x000